Algorithms for Programming Contests - Week 3

Stefan Jaax, Philipp Meyer, Christian Müller
conpra@in.tum.de

11.05.2017

Graphs

A *graph* is a tuple $G = (V, E)$, where V is a non-empty set of *vertices* and E is a set of edges.
A *directed* graph is a graph with $E \subseteq V \times V = \{(u, v) \mid u, v \in V\}$.
An *undirected* graphs is a graph with $E \subseteq \{\{u, v\} \mid u, v \in V\}$.

For a vertex v, we denote the successors of v by $vE := \{u \mid (v, u) \in E\}$ for directed graphs $vE := \{u \mid \{v, u\} \in E\}$ for undirected graphs.

A *path* from v_1 to v_n is a sequence $p = v_1 v_2 \ldots v_n$ such that $v_{i+1} \in v_i E$ for all $i \in [1, n-1]$, and $v_i \neq v_j$ for all $i \neq j$.

- A graph is *cyclic* if there is a path $p = v_1 \dots v_n$ with $v_1 \in v_n E$, otherwise it is *acyclic*.
- An undirected graph is *connected* if for every pair of vertices $u, v \in V$, there is path from u to v.
- For an undirected graph, a *connected component* is a maximal set $V' \subseteq V$ where for all $u, v \in V'$, there is a path from u to v.
- An undirected graph is a *tree* if it is acyclic and connected. For any tree (V, E), we have $|V| = |E| + 1$.
- An undirected acyclic graph is a *forest*, and each connected component is a tree.
- A directed acyclic graph is also called a *DAG*.

Graphs as an abstract data type

Graph representation

- Adjacency list: For each vertex v, store a list of successors vE.
- Adjacency matrix: For each pair of vertices u, v, store existence of an edge $(u, v) \in E$.

Graph operations

- Make graph: build a graph from a list of vertices and edges.
- Get vertices: Iterate over alle vertices $v \in V$.
- Get edges: Iterate over alle edges $e \in E$.
- Test edge: Test existence of an edge $(u, v) \in E$.
- Get successors: For a vertex v, iterate over alle successors $u \in vE$.

Graph traversal

Graph traversal

- Visit vertices in certain order.
- Assign vertices an order $o : V \to \mathbb{N} \cup \{\infty\}$ of discovery time.
- Possibly keep track of other information such as finishing time, predecessor, etc.

Usages

- Find vertex with certain properties.
- Check property for all vertices.
- Find connected components.
- Check for cycles.
- ...

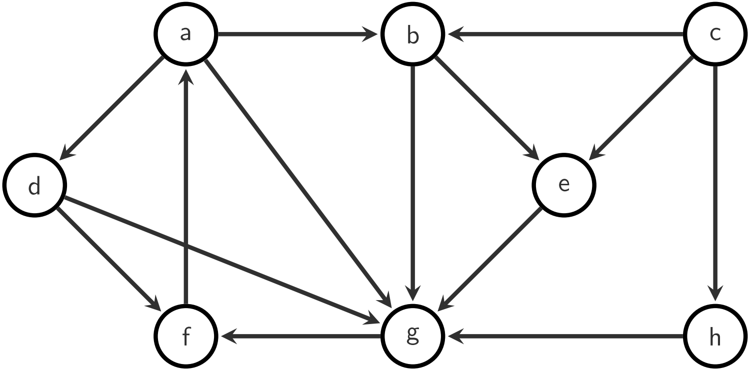
Depth-first search (DFS)

Algorithm 1 Depth-first search

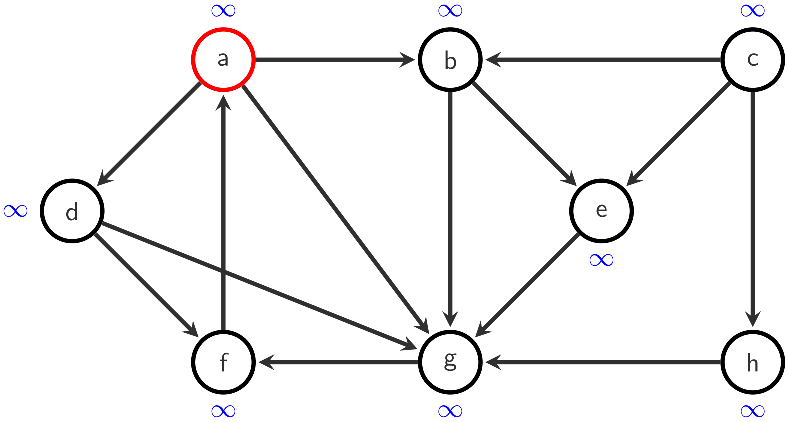
```
Input: Graph G = (V, E)
  procedure DFS(G)
      for each vertex v ∈ V do
          o(v) ← ∞
      end for
      S ← EmptyStack()
      i ← 1
      for each vertex v ∈ V do
          if o(v) = ∞ then
              DFSExplore(G, v)
          end if
      end for
  end procedure

procedure DFSExplore(G, v)
    S.push(v)
    while S is not empty do
        v = S.pop()
        if o(v) = ∞ then
            o(v) ← i;
            i ← i + 1
            for each u ∈ vE do
                S.push(u)
            end for
        end if
    end while
end procedure
```

DFS (example)

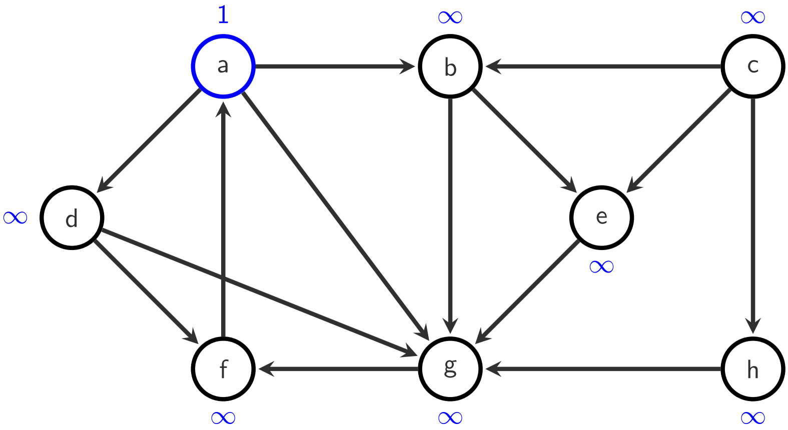


DFS (example)



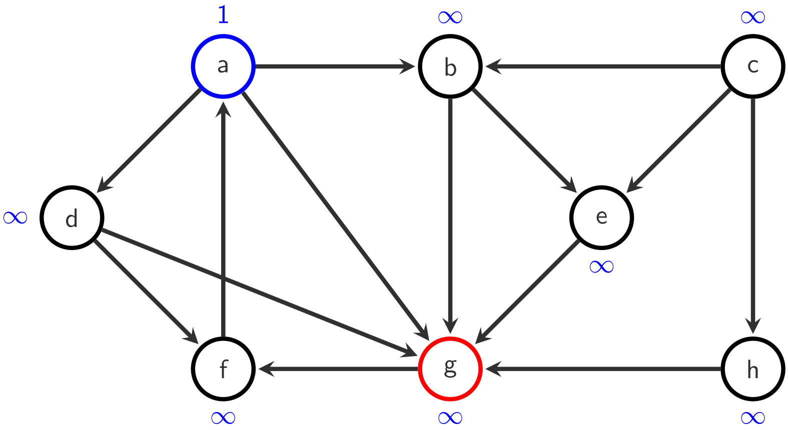
$S = [a]$

DFS (example)



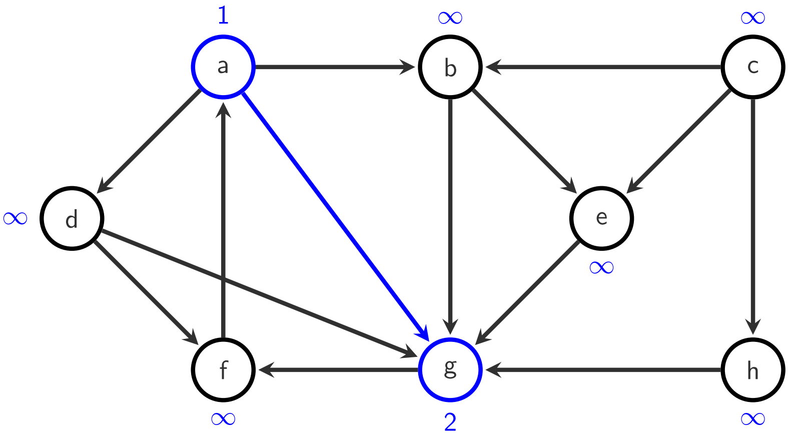
$S = [\mathrm{b}, \mathrm{d}, \mathrm{g}]$

DFS (example)



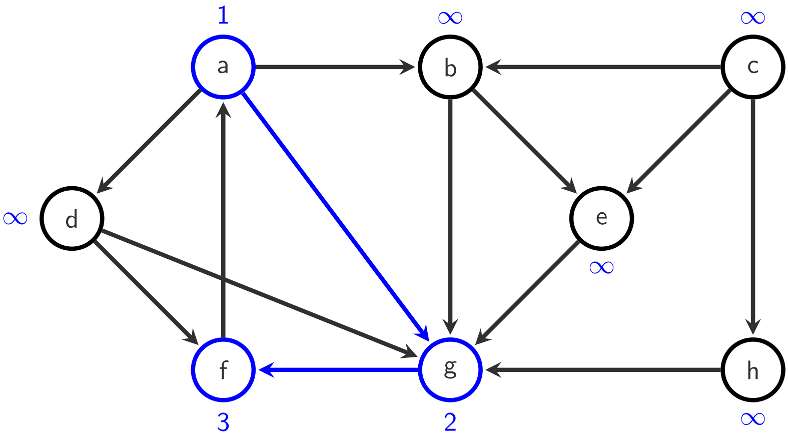
$S = [\text{b}, \text{d}, \text{g}]$

DFS (example)



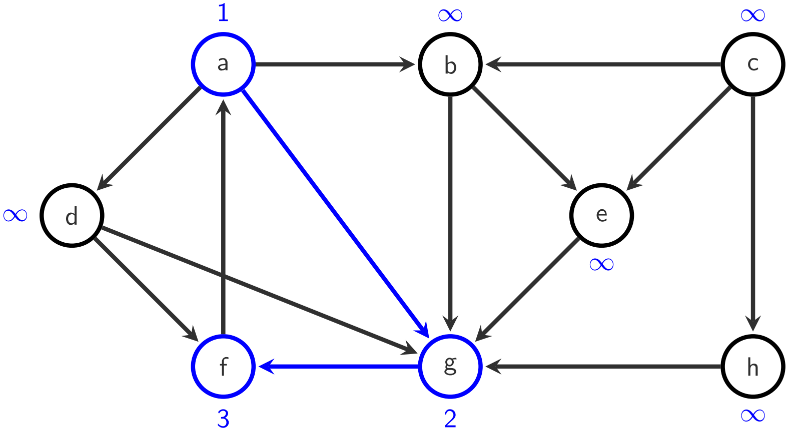
$S = [\text{b}, \text{d}, \text{f}]$

DFS (example)



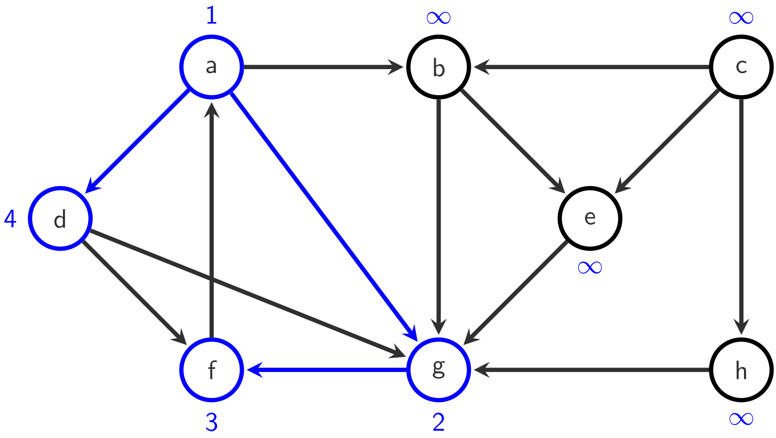
$S = [\mathrm{b}, \mathrm{d}, \mathrm{a}]$

DFS (example)



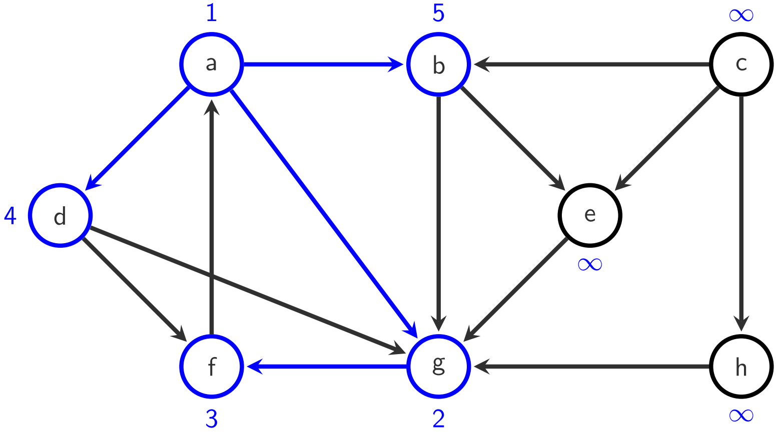
$S = [\mathrm{b}, \mathrm{d}]$

DFS (example)



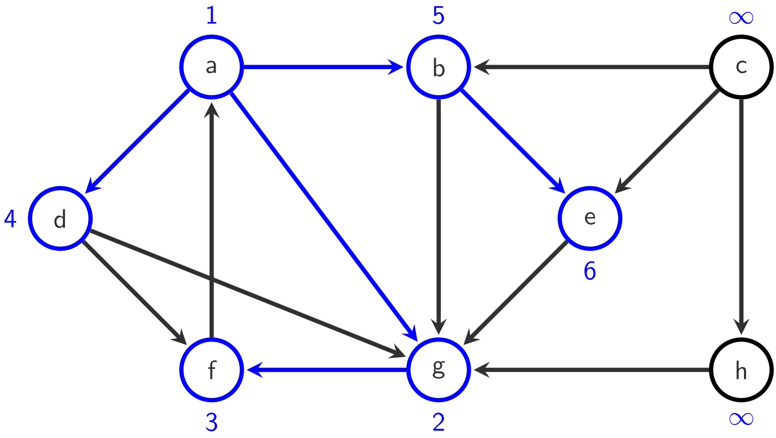
$S = [\mathrm{b}, \mathrm{f}, \mathrm{g}]$

DFS (example)



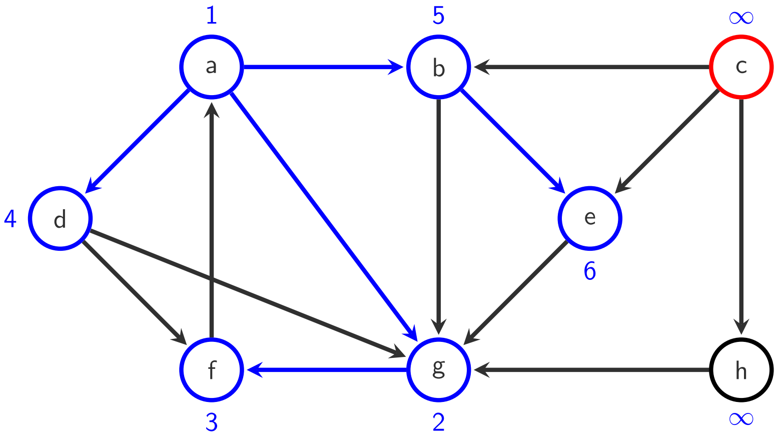
$S = [\mathrm{e}, \mathrm{g}]$

DFS (example)



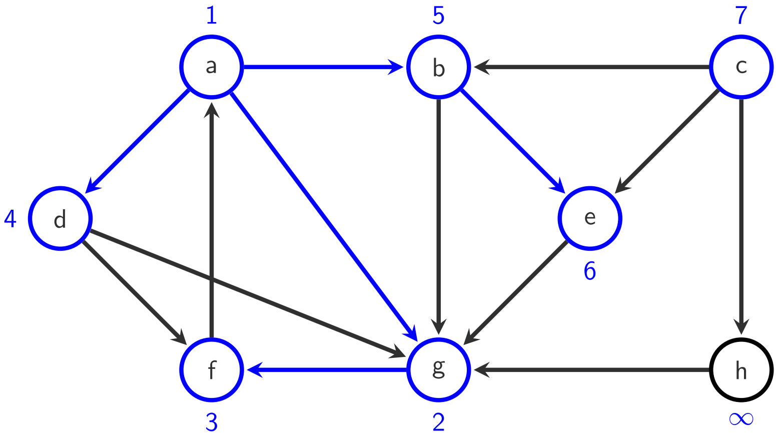
$S = [\text{g}]$

DFS (example)



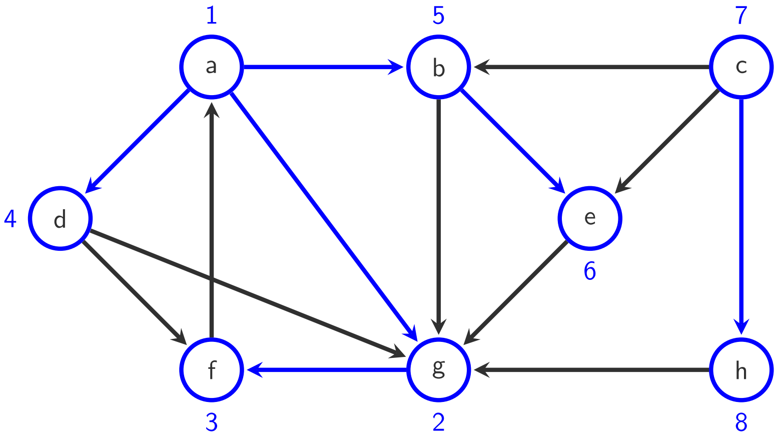
$S = [c]$

DFS (example)



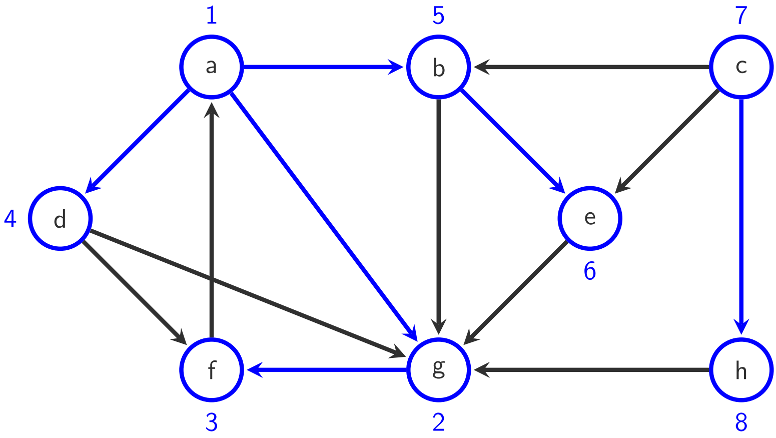
$S = [\mathrm{e}, \mathrm{h}]$

DFS (example)



$S = [\mathrm{e}, \mathrm{g}]$

DFS (example)



$S = []$

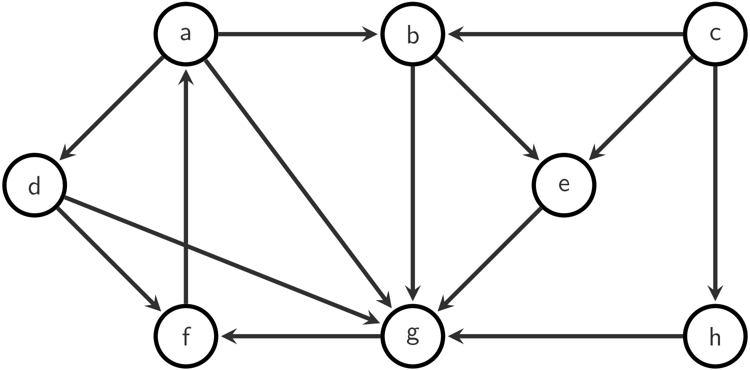
Breadth-first search (BFS)

Algorithm 2 Breadth-first search

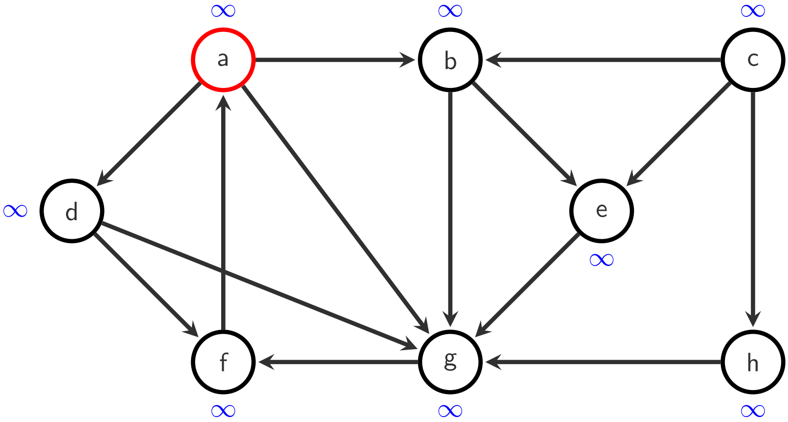
```
Input: Graph G = (V, E)
  procedure BFS(G)
      for each vertex v ∈ V do
          o(v) ← ∞
      end for
      S ← EmptyQueue()
      i ← 1
      for each vertex v ∈ V do
          if o(v) = ∞ then
              BFSExplore(G, v)
          end if
      end for
  end procedure

procedure BFSExplore(G, v)
    S.enqueue(v)
    while S is not empty do
        v = S.dequeue()
        o(v) ← i;
        i ← i + 1
        for each u ∈ vE do
            if u(v) = ∞ then
                o(v) ← 0;
                S.enqueue(u)
            end if
        end for
    end while
end procedure
```

BFS (example)

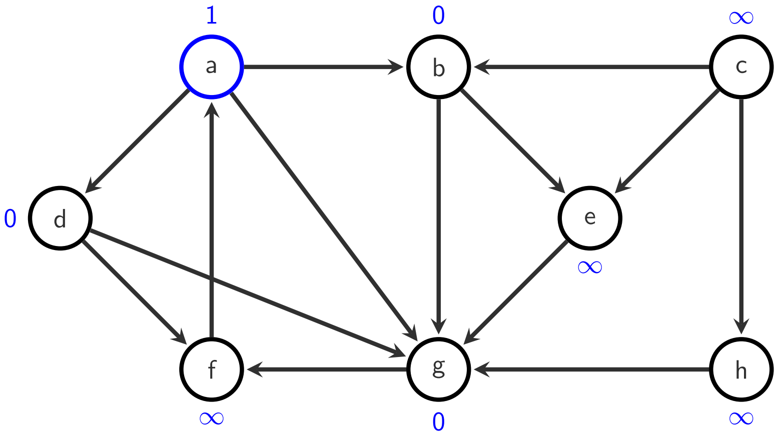


BFS (example)



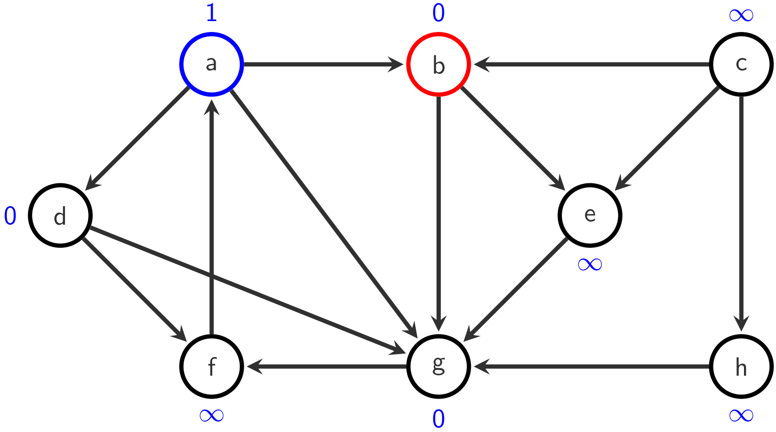
$S = [a]$

BFS (example)



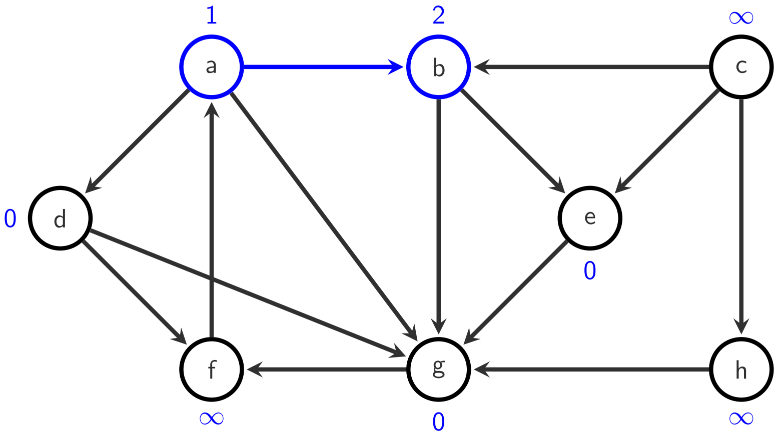
$S = [\text{b}, \text{d}, \text{g}]$

BFS (example)



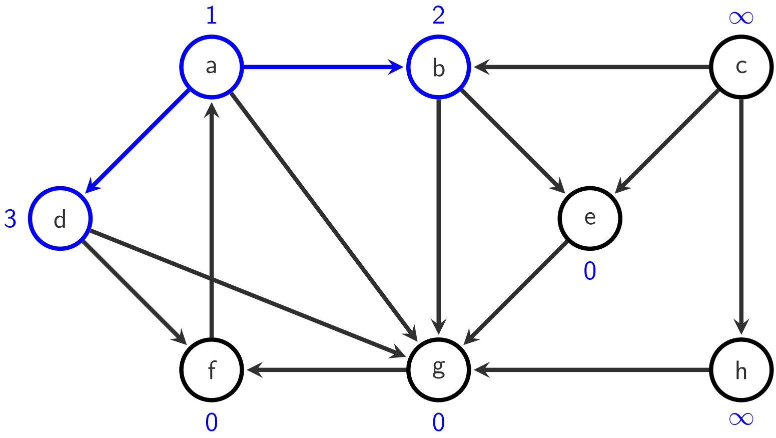
$S = [\textcolor{red}{b}, d, g]$

BFS (example)



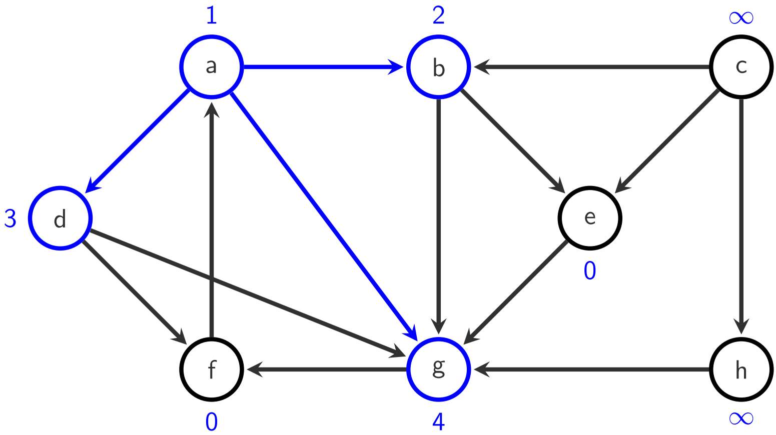
$S = [\mathrm{d}, \mathrm{g}, \mathrm{e}]$

BFS (example)



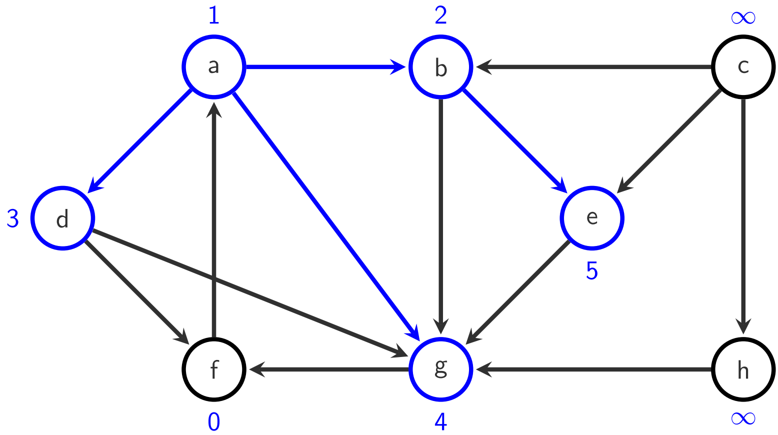
$S = [\mathrm{g}, \mathrm{e}, \mathrm{f}]$

BFS (example)



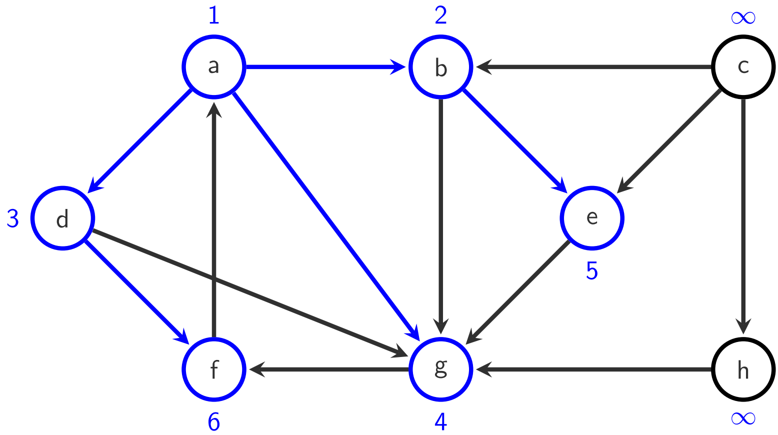
$S = [\mathrm{e}, \mathrm{f}]$

BFS (example)



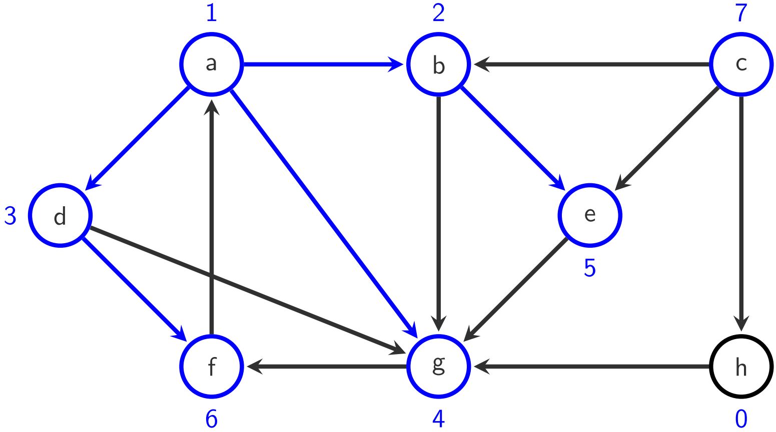
$S = [\text{f}]$

BFS (example)



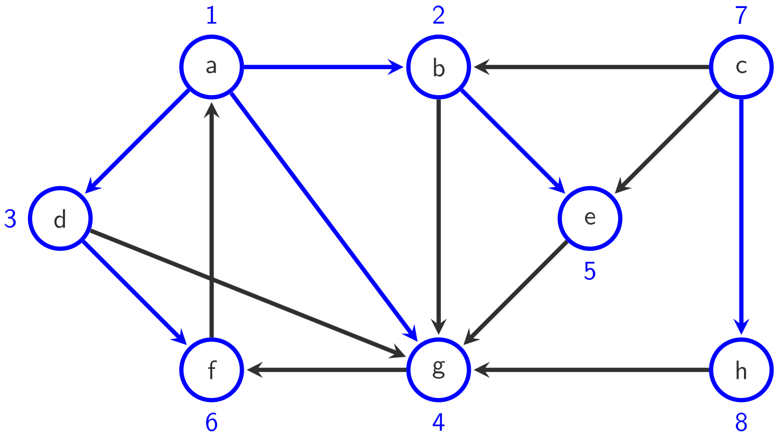
$S = []$

BFS (example)



$S = [\mathrm{h}]$

BFS (example)



$S = []$

Topological sort (TS)

Topological order

For a directed graph $G = (V, E)$, a *topological order* is an assignment $o : V \to \mathbb{N}$ such that for all $(u, v) \in E$, we have $o(u) < o(v)$.

- Topological order exists *if and only if* graph is acyclic (i.e. a DAG).
- Topological order may not be unique.
- Topological sort: Problem of finding a topological order.

Usages

- Resolving dependencies.
- Instruction scheduling.
- Determine order for compilation multi-source programs.
- Detecting cycles.

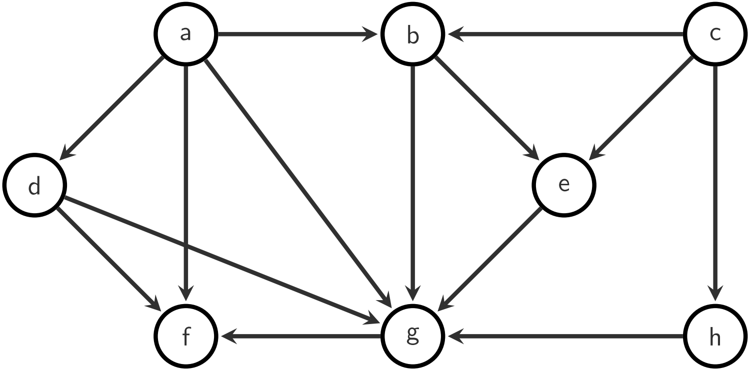
Algorithm 3 Topological sort

Input: Directed graph $G = (V, E)$
procedure $\mathrm{TS}(G)$
  for each vertex $v \in V$ **do**
    $o(v) \leftarrow \infty$
    ▷ count predecessors
    $pre(v) \leftarrow |\{u \mid v \in uE\}|$
  end for
  $S \leftarrow$ EmptyStack()
  $i \leftarrow 1$
  for each vertex $v \in V$ **do**
    if $pre(v) = 0$ **then**
      $\textsc{TSExplore}(G, v)$
    end if
  end for
end procedure

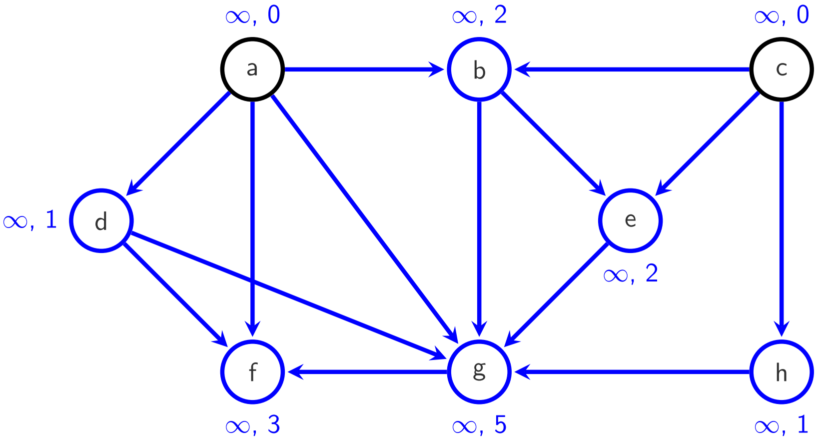
procedure $\textsc{TSExplore}(G, v)$
  S.push(v)
  while S is not empty **do**
    $v = S$.pop()
    $o(v) \leftarrow i$; $i \leftarrow i + 1$
    for each $u \in vE$ **do**
      $pre(u) \leftarrow pre(u) - 1$
      if $pre(u) = 0$ **then**
        S.push(u)
      end if
    end for
  end while
end procedure

If vertices with $o(v) < 0$ remain, the graph is cyclic.

TS (example)

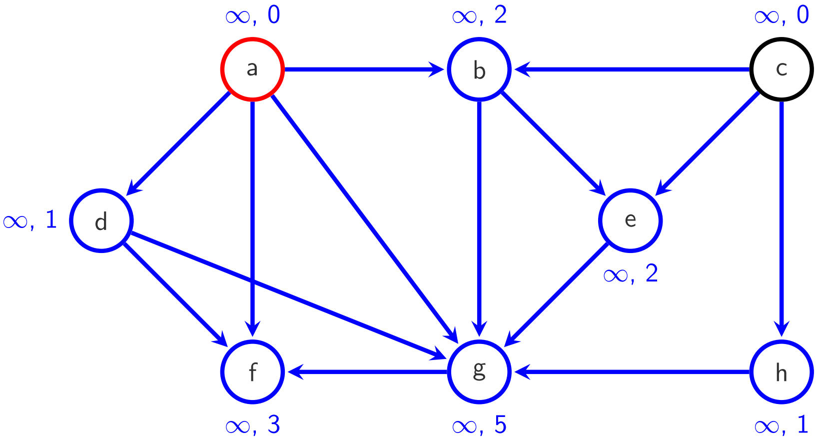


TS (example)



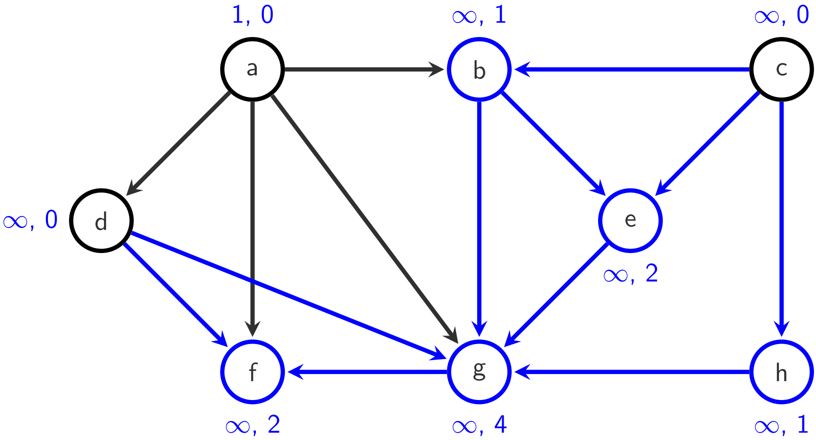
$S = [\mathrm{a}, \mathrm{c}]$

TS (example)



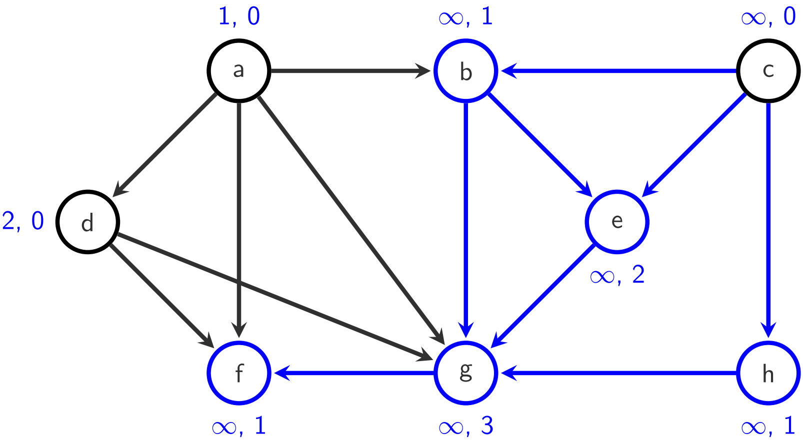
$S = [a, c]$

TS (example)



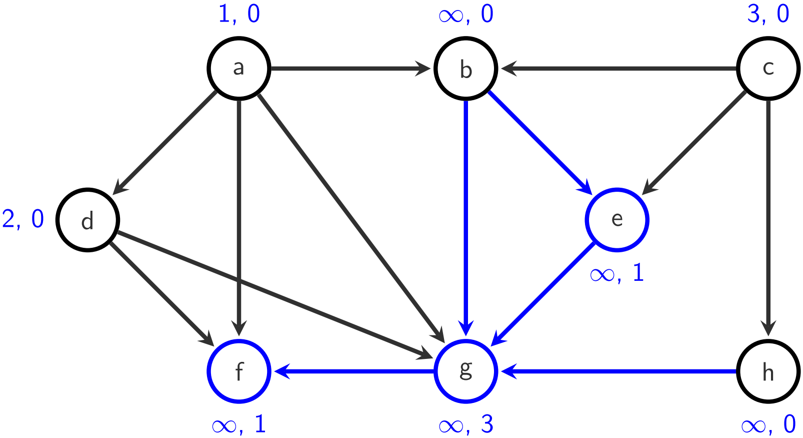
$S = [\mathrm{c}, \mathrm{d}]$

TS (example)



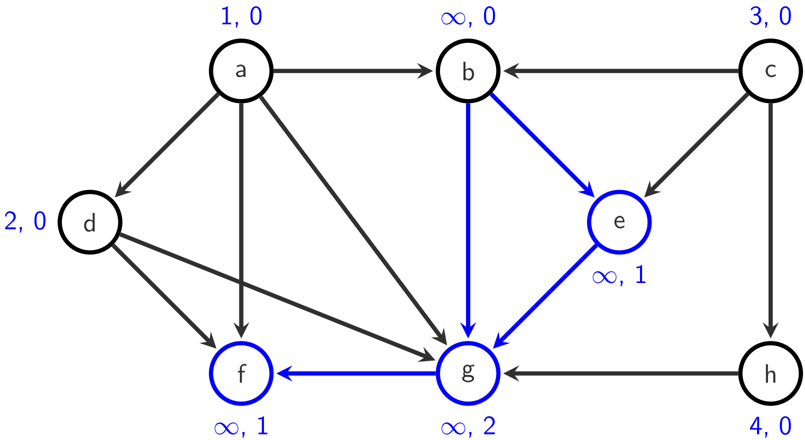
$S = [\mathrm{c}]$

TS (example)



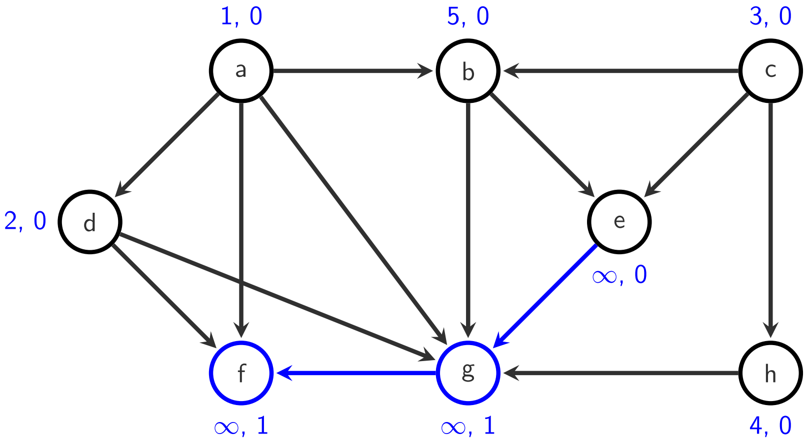
$S = [\mathrm{b}, \mathrm{h}]$

TS (example)



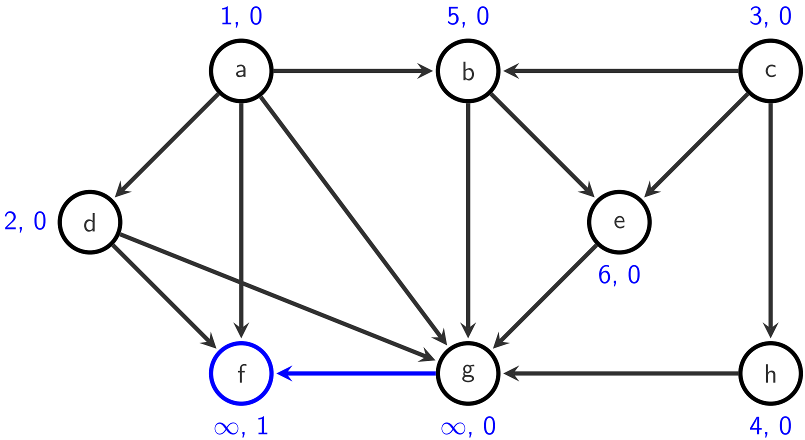
$S = [\text{b}]$

TS (example)



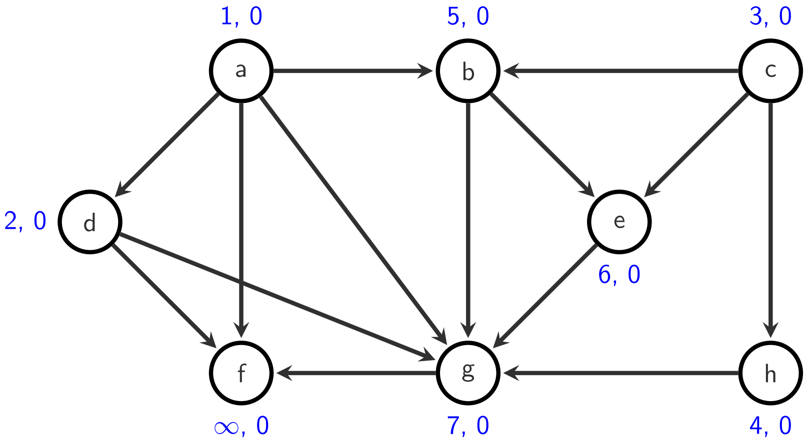
$S = [\text{e}]$

TS (example)



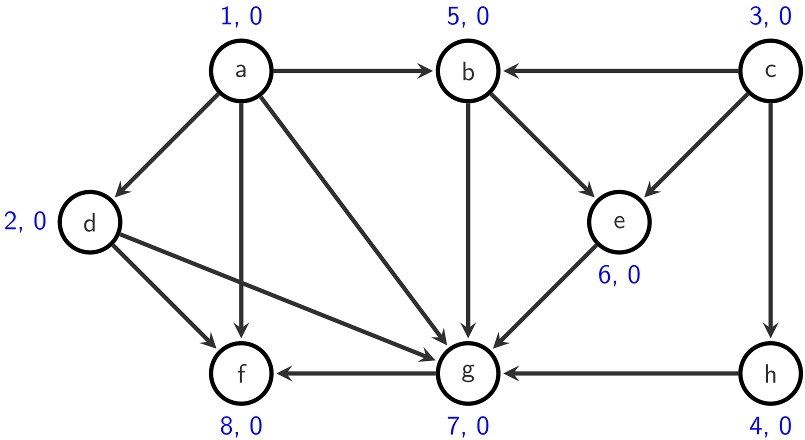
$S = [\text{g}]$

TS (example)



$S = [\text{f}]$

TS (example)



$S = []$

Analysis of DFS, BFS and TS

Running time

- Each vertex is visited at most once: $\mathcal{O}(|V|)$
- For each vertex, each successor considered at most once: $\mathcal{O}\left(\sum_{v \in V} |vE|\right) = \mathcal{O}(|E|)$
- In total: $\mathcal{O}(|V| + |E|)$
- For topological sort, count number of predecessors in linear time.

Minimum spanning trees (MST)

Spanning tree

For an undirected graph $G = (V, E)$, a *spanning tree* of G is a subset of edges $T \subseteq E$ such that (V, T) forms a tree, i.e. is connected and acyclic.

Weighted graphs

We now consider graphs with a *weight function* $w : E \to \mathbb{R}$ on the edges. For a subset of edges $E' \subseteq E$, we define $w(E') := \sum_{e \in E'} w(e)$.

Minimum (weight) spanning tree

For an undirected graph $G = (V, E)$ with a weight function $w : E \to R$, a *minimum spanning tree (MST)* is a spanning tree S of G such that for all spanning trees T of G, we have $w(S) \leq w(T)$.

Minimum spanning trees (MST)

- Spanning trees only exist for connected graphs.
- Otherwise, a spanning tree exists for each connected component.
- All spanning trees of a graph have the same number of edges.
- Negative weights can be avoided by adding a constant to all weights.
- Maximum spanning tree can be obtained with $w'(e) = -w(e)$.

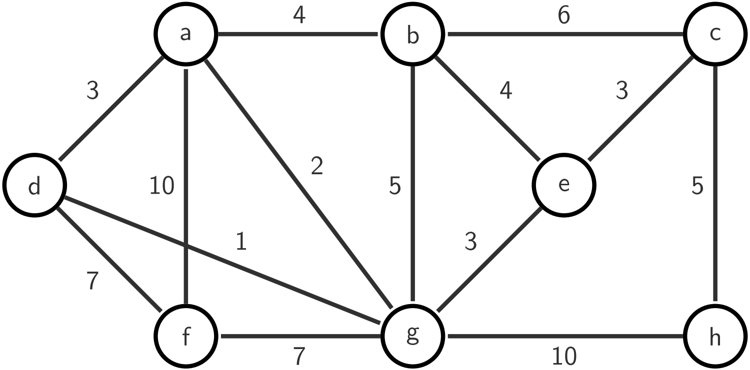
Algorithm 4 Kruskal's algorithm

Input: Undirected graph $G = (V, E)$

procedure $\textsc{Kruskal}(G)$
    $S \leftarrow \emptyset$ ▷ Current set of edges
    $L \leftarrow$ List of edges $e \in E$ sorted in increasing order by $w(e)$
    $U \leftarrow$ Union-Find structure initialized over set V
    for each edge (u, v) in L in order **do**
        ▷ Test if vertices are in different components
        if $U.\text{find}(u) \neq U.\text{find}(v)$ **then**
            ▷ If yes, add edge to MST and merge components
            $U.\text{union}(u, v)$
            $S \leftarrow S \cup \{e\}$
        end if
    end for
end procedure

If vertices in different components remain, the graph is not connected.

Kruskal (example)



Analysis of Kruskal's algorithm

Running time

- Sorting of edges: $\mathcal{O}(|E| \log |E|)$
- With α as the inverse Ackermann function, i.e. $\alpha = f^{-1}$ with $f(n) = A(n, n)$:
- $2|E|$ find operations: $\mathcal{O}(|E| \alpha(|V|))$
- $|V|$ union operations: $\mathcal{O}(|V| \alpha(|V|))$
- In total: $\mathcal{O}(|E| \log |E|)$

Proof of correctness for Kruskal's algorithm

Lemma

Let $T \subseteq E$ be a set of edges such that there is a minimum spanning tree S of G with $E' \subseteq S$.
Let $e \in E \setminus T$ be an edge such that $T \cup \{e\}$ does not create a cycle, with e having minimal weight among all of these edges.
Then, there is a minimum spanning tree S' of G such that $T \cup \{e\} \subseteq S'$.

Proof.

When $e \in S$, then $S' := S$ fulfills the requirement.
When $e \notin S$, then $S \cup \{e\}$ has a cycle c, and there is an edge $f \neq e$ in c that is not in T (otherwise adding e to T would create a cycle). Then $S' := S \setminus \{f\} \cup \{e\}$ is a also a spanning tree, and $w(S') \leq w(S)$, as $w(e) \leq w(f)$. As S is a minimum spanning tree, we have $w(S') = w(S)$, and therefore S' is also a minimium spanning tree.

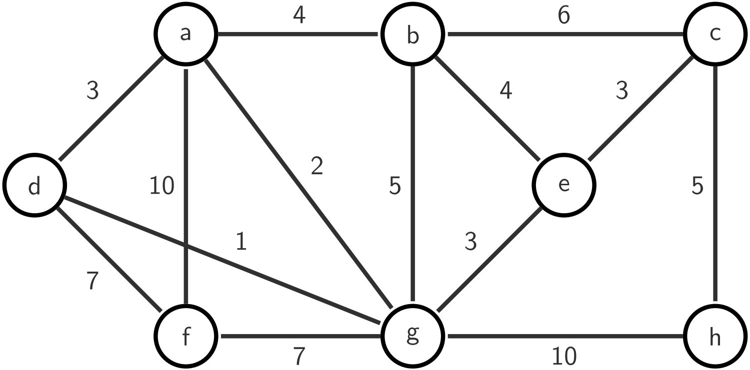
Algorithm 5 Prim's algorithm

```
Input: Graph G = (V, E)
  procedure PRIM(G)
    S ← ∅
    for each vertex v ∈ V do
      visited(v) ← false
      c(v) ← ∞
    end for
    PQ ← PriorityQueue over V
    s ← any v ∈ V
    PRIMVISIT(s)
    while P is not empty do
      v ← PQ.deleteMin()
      S ← S ∪ {{pre(v), v}}
      PRIMVISIT(v)
    end while
  end procedure

  procedure PRIMVISIT(v)
    visited(v) ← true
    for each u ∈ vE do
      if not visited(w) then
        if w(v, u) < c(u) then
          pre(u) ← v
          c(u) ← w(v, u)
          if w in PQ then
            PQ.decreaseKey(u, c(u))
          else
            PQ.insert(u, c(u))
          end if
        end if
      end if
    end for
  end procedure
```

If not all vertices were visited, the graph is not connected.

Prim (example)



Analysis of Prim's algorithm

Running time

- Graph exploration without priority queue: $\mathcal{O}(|V| + |E|)$
- With Fibonacci heap as priority queue:
- $|V|$ insert operations: $\mathcal{O}(|V|)$
- $|E|$ decreaseKey operations: $\mathcal{O}(|E|)$
- $|V|$ deleteMin operations: $\mathcal{O}(|V| \log |V|)$
- In total: $\mathcal{O}(|E| + |V| \log |V|)$

Note: In Java and C++, there is no decreaseKey operation, instead delete and insert again.